Directions: Submit one pdf file only, put your name (last, first) as the name of the file. Inside the file also put your name (Last, First) and N number. Keep your answers in order in your file 1, 2 … 1) (20 pts)

1. Consider the following program written in C syntax:

void swap (int a, int b)

{ int temp;

temp = a;

a = b;

b = temp; } void main() { int value = 2, list[5] = {1, 3, 5, 7, 9}; swap(value, list[0]); swap(list[0], list[1]); swap(value, list[value]); } For each of the following parameter-passing methods, what are all of the values of the variables value and list after each of the three calls to swap? a) (10 pts)

**Passed by reference**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Value | List[0] | List[1] | List[2] | List[3] | List[4] | Logic |
| 2 | 1 | 3 | 5 | 7 | 9 | Original Values |
| 1 | 2 | 3 | 5 | 7 | 9 | a:move value address in  b:move list[0] address in  temp = a  a=b  b=temp  Values Swapped |
| 1 | 3 | 2 | 5 | 7 | 9 | a:move list[0] address in  b:move list[1] address in  temp = a  a=b  b=temp  Values Swapped |
| 2 | 3 | 1 | 5 | 7 | 9 | a:move value address in  b:move list[value] address in  temp = a  a=b  b=temp  Values Swapped |

**Passed by Value-Result**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Value | List[0] | List[1] | List[2] | List[3] | List[4] | Logic |
| 2 | 1 | 3 | 5 | 7 | 9 | Original Values |
| 1 | 2 | 3 | 5 | 7 | 9 | Addr\_value:move value address in  Addr\_list[0]: move list[0] address in  a:move Addr\_value value in  b:move Addr\_list[0] value in  temp = a  a=b  b=temp  Values Swapped  Addr\_Value = move first para value out  Addr\_list[0] = move second para value out |
| 1 | 3 | 2 | 5 | 7 | 9 | Addr\_list[0]:move list[0] address in  Addr\_list[1]: move list[1] address in  a:move Addr\_list[0] value in  b:move Addr\_list[1] value in  temp = a  a=b  b=temp  Values Swapped  Addr\_list[0] = move first para value out  Addr\_list[1] = move second para value out |
| 2 | 3 | 1 | 5 | 7 | 9 | Addr\_value:move value address in  Addr\_list[value]: move list[value] address in  a:move Addr\_value value in  b:move Addr\_list[value] value in  temp = a  a=b  b=temp  Values Swapped  Addr\_value = move first para value out  Addr\_list[value] = move second para value out |

2. (20 pts) What does the following program do. Show the output. Explain.

#include <stdio.h>

#include <setjmp.h>

jmp\_buf bufferA, bufferB;

void routineB();

void routineA() {

int r ;

printf("(A1)\n");

r = setjmp(bufferA);

if (r == 0) routineB();

printf("(A2) r=%d\n",r);

r = setjmp(bufferA);

if (r == 0) longjmp(bufferB, 20001);

printf("(A3) r=%d\n",r);

r = setjmp(bufferA);

if (r == 0) longjmp(bufferB, 20002);

printf("(A4) r=%d\n",r);

}

void routineB() {

int r;

printf("(B1)\n");

r = setjmp(bufferB);

if (r == 0) longjmp(bufferA, 10001);

printf("(B2) r=%d\n", r);

r = setjmp(bufferB);

if (r == 0) longjmp(bufferA, 10002);

printf("(B3) r=%d\n", r);

r = setjmp(bufferB);

if (r == 0) longjmp(bufferA, 10003);

}

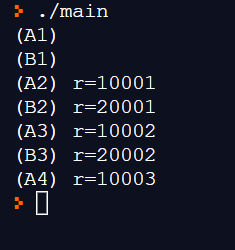
int main(int argc, char \*\*argv) {

routineA();

return 0;

}

Output:



|  |  |
| --- | --- |
| S.No | Execution and Control Flow |
| 1 | Main Program Calls routineA and **prints "(A1)"** |
| 2 | The first Setjmp assigns 0 to r with env= "bufferA", condtional statement evaluates to 0 and calls routineB |
| 3 | routineB exectuion begins and **prints "(B1)"** |
| 4 | routineB's first setjump assings 0 to r,condtional statement evaluates to 0 and longjmp in condtional statement takes the control back to first setjmp(bufferA) under routine A |
| 5 | The value 10001 is assigned to r and The condtional statement now **prints "(A2) r=10001"** as r is not equal to 0 |
| 6 | r is again reset to 0 on routineA's second setjmp with env = "bufferA" |
| 7 | In next condtional statement r equates to 0 and longjmp moves control to first setjmp in routine B with value of 20001 assigned to r |
| 8 | The condtional statement evaluates to r not equals 0 and **prints** **"(B2) r=20001"** |
| 9 | r is again reset to 0 with env = "BufferB" on routineB's second setjmp, The condtional statement evalues to 0 and longjmp returns control to 2nd setjmp in rotuine A with value of 10002 |
| 10 | The condtional statement evaluates to r not equals 0 and **prints" (A3) r=10002"** |
| 11 | r is again reset to 0 on routineA's third setjmp and longjmp returns control to 2nd setjmp in rotuine B with value of 20002 |
| 12 | The condtional statement evaluates to r not equals 0 and prints **"(B3) r=20002"** |
| 13 | r is again reset to 0 on routineB's third setjmp and longjmp returns control to 3rd setjmp in rotuineA with value of 10003 |
| 14 | The condtional statement evaluates to r not equals 0 and prints **"(A4) r=10003"** |

1. Show the ARI assuming execution reaches position 1 with deep-access dynamic method

The sequence of procedure calls is

Main calls fun2

fun2 calls fun1

fun1 calls fun1

fun1 calls fun3

|  |  |
| --- | --- |
| ARI for fun3 | Local - d |
| Dynamic Link |
| Return to fun1 |
| ARI for fun1 | Local - a |
| Dynamic Link |
| Return to fun1 |
| ARI for fun1 | Local - a |
| Dynamic Link |
| Return to fun2 |
| ARI for fun2 | Local - c |
| Local - b |
| Dynamic Link |
| Return to Main |
| ARI for Main | Local -g |
| Local -f |
| Local -e |

4) (10 pts) Write a function in Prolog that takes two lists of integers and returns a list containing only those elements that are unique to both. i.e. func([1,2,3], [2, 4, 6]) -> [2]

% Element Y is in list?

present(Y, [ Y | \_ ]).

present(Y, [ \_ | L ]):- present(Y, L).

% intersection of two list

inter([ ], \_, [ ]).

inter([ X | A ], B, [ X | C ]):- present(X, B), inter(A, B, C).

inter([ \_ | A ], B, C):- inter(A, B, C).

5) (10 pts) Write a function in Julia that takes two lists of integers and returns a list containing only those elements that are unique to both. i.e. func([1,2,3], [2, 4, 6]) -> [2]

function intersect(lst1,lst2)

if lst1 == []

[]

elseif lst2 == []

[]

elseif first(lst1) in lst2

append!(intersect(lst1[2:size(lst1)[1]],deleteat!(lst2, findall(x->x==first(lst1),lst1))),first(lst1))

else

intersect(lst1[2:size(lst1)[1]],lst2)

end

end

6) (10 pts) Write a function in Haskell that takes two lists of integers and returns a list containing only those elements that are unique to both. i.e. func([1,2,3], [2, 4, 6]) -> [2]

filterNot f = filter (not . f)

intersect lst1 lst2

| lst1 == [] = []

| lst2 == [] = []

| elem (head lst1) lst2 = (head lst1):intersect (tail lst1) (filterNot (==(head lst1)) lst2)

| otherwise = intersect (tail lst1) lst2

7) (10 pts) Describe the main features of Aspect Oriented Programming (AOP). Show examples from 3 languages that support AOP in part or in whole. Talk about the benefits of AOP as well as any shortcomings that may exist.

Aspect oriented programming(AOP) uses aspects in programming. It can be defined as the breaking of code into different modules where the aspect is the key unit of modularity. Aspects enable the implementation of crosscutting concerns such as- transaction, logging not central to business logic without cluttering the code core to its functionality. It does so by adding additional behavior that is the advice to the existing code. For example- Security is a crosscutting concern, in many methods in an application security rules can be applied, therefore repeating the code at every method, define the functionality in a common class and control were to apply that functionality in the whole application.

Main features of AOP:

1. Aspect: corresponds to “What to be added” .These are the behaviors/methods we are looking to inject into the target code. In simpler words, this is the new logic to be added to the existing class, method, sets of classes, or sets of methods.
2. Advice: specify some common moments when aspect’s code needs to be executed, such as “before”, “after”, “around”, “whenThrowing”, and the like. Advice, refer to the moment in time-related to the execution of the code. For the ones referring to after the code is executed, the aspects will intercept the returned value and potentially overwrite it if they needed to.
3. Pointcut: references the place in the target code where the aspect needs to be injected.
4. Joint point: A point through the program implementation as in the method execution or the management of an exception
5. cross-cut : Concerns tangled and scattered across many modules in a program.

Example: is adding log messages to the execution of certain functions.

**Core Program:**

@Component

class PerformACommand {

@Logged

fun execute(input: String): String {

return "this is a result for $input"

}

}

**AOP program**

**Aspect:**

@Aspect

@Component

class LoggingAspect {

    @Around("@annotation(Logged)")

    fun logMethod(joinPoint: ProceedingJoinPoint) {

        var output = joinPoint.proceed()

        println("method '${joinPoint.signature}' was called with input '${joinPoint.args.first()}' and output '$output'")

    }

}

**JoinPoint:**

@Target(AnnotationTarget.FUNCTION)

annotation class Logged

**PointCut:**

@Around("@annotation(Logged)")

2. C: "Hello World" program:

**Advise:**

**Core Program**

int main() {

printf("Hello ");

}

**the aspect program is:**

after(): execution(int main()) {

printf(" World from !AspeCt-oriented C ! \n");

}

**Output is:**

Hello World from !AspeCt-oriented C !

Benefits of Aspect Oriented Programming:

1. Quick Code Reuse

Adding aspects allows for reuse of code across many, many classes. Example a simple annotation like “Logged,” can be used to enhance numerous classes without repeating exact logging logic.

1. Dealing With Third-Party Code

With AOP, its easy to alter the third party code. By just decorate the needed behavior without touching the third-party code at all needed functionality can be introduced. Example: Tracer to track the control /execution flow.

1. Cross-Cutting Concerns allows for efficient Code maintenance

Cross-cutting concerns are parts of a program that rely on or must affect many other parts of the system. AOP allows for surgically slicing out the pieces/aspects of the core components that aren’t connected to its main behavior: authentication, logging, tracing, error handling, and the like. As a result. The core components will be much more readable and changeable as a result and the cross-cutting concerns can be easily reused and managed.